

29

Octobre 1986

EDITORIAL

Dans le numéro d'octobre du mensuel MICRO-SYSTEMES, vous trouverez un article, qui, si vous ne l'avez déja lu, vous séduira: "chaos fractal sur AMSTRAD". On pourrait critiquer le fait qu'il soit écrit en BASIC, mais pour une fois passons ce détail. Voici l'exemple le plus merveilleux d'utilisation d'un micro-ordinateur familial (et à vocation semi-pro...) pour la plus inutile des activités: un programme qui ne sert à rien, un programme qui existe comme existe un tableau ou une musique. De la complexité des algorithmes se dégage la beauté formelle et impalpable du sens caché de la nature des nombres. Le ballet des chiffres se déplaçant sur une scène à quatre dimensions ne nous laissent voir que l'ombre de leur existence.

Et ce préambule me remémore un ouvrage que j'avais lu et apprécié, "CONTACT" écrit par Carl SAGAN. Le thème du livre est simple; on réussit à capter des signaux provenant d'une intelligence extra-terrestre. Mais dans ce livre, il est aussi question de chiffres. On commence par les nombres premiers, puis... (mais je ne dévoilerai pas l'histoire). Et nombre de savants se mettent à chercher les clefs du message, car il y a un message et même les plans d'une machine. Dans ce livre, Carl SAGAN mêle avec habileté la politique, la religion, l'informatique et les mathématiques (encore...). La grande surprise vous est assénée à la fin du livre, car après un fantastique voyage à l'autre bout de l'univers, vous ne saurez toujours pas à quoi ressemblent les extra-terrestres, mais vous saurez que eux aussi cherchent la clef du message, celui contenu dans un nombre, le nombre le plus répandu dans l'univers...

Peut-être Carl SAGAN a-t-il raison, mais la conclusion de son livre reste quand même une hypothèse de travail. Toujours est-il que je n'ai pu m'empêcher de faire un rapprochement entre cette histoire de clef contenue dans un nombre et les images fractales.

Alors peut-être doit-on considérer les ordinateurs comme des microscopes dont la destination essentielle est la dissection d'entités non physiques restant encore à découvrir. Autrefois, on recherchait la clef de la vie au bout d'une lentille et l'on distinguait à peine les bactéries. Depuis, avec le microscope électronique, on atteint la molécule, mais on n'a pas encore trouvé la vie. Nos ordinateurs actuels, si puissant soient-ils, semblent encore aussi primitifs que la première lentille de grossissement au regard des entités mathématiques à manipuler. Les plus grands centres de calcul s'enfoncent avec peine dans des domaines de recherche où la connaissance théorique construit des modèles d'univers dont les fractals n'en sont qu'un des exemples les plus simples.

Cette notion de calcul est certes fondamentale, mais les idées aussi ont leur importance. Chaque nouveau modèle mathématique peut avoir des retombées pratiques ou philosophiques. La géométrie fractale chasse celle d'Euclide comme la physique Einsteinienne a écrasé celle de Newton (...et l'Amstrad le ZX 81..). Et puis, quel merveilleux terrain de recherche. Il suffit de disposer d'un stylo, de papier et d'un ordinateur.

SOMMAIRE

Toute reproduction, adaptation, traduction partielle du contenu de ce magazine, sous toutes les formes est vivement encouragée, à l'exclusion de toute reproduction à des fins commerciales. Dans le cas de reproduction par photocopie, il est demandé de ne pas masquer les références inscrites en bas de page, et dans les autres cas, de citer l'ASSOCIATION JEDI. Pour tout renseignement, vous pouvez nous contacter en nous écrivant à l'adresse suivante:

ASSOCIATION JEDI 8, rue Poirier de Narçay 75014 PARIS

Tel: (1) 45.42.88.90 (de 10h à 18h)

* LE PHRASE EN FORTH *

par W. Baden et K. Clark, traduction A. J., aout 86.

Phrasé: (n.m.) Manière, art de phraser, en musique. (Petit Robert).
"Le phrasé est un art subjectif; je n'ai jamais trouvé un ensemble utilisable de règles formelles."
(Leo Brodie, "Débutez en Forth", ed. Eyrolles)

De quoi s'agit-il?

Le phrasé, en Forth, se réfère à la façon de placer des espaces supplémentaires entre les mots dans les définitions. Ce qui groupe le code en "phrases" permettant au lecteur de le comprendre. Le "phrasé" aide l'humain, mais pas le calculateur dans l'analyse du code.

L'analyse grammaticale consiste à identifier chaque mot de chaque partie d'une phrase. Un moyen effectif d'analyse des phrases Forth est de revenir à ce sens premier, c.à.d. d'identifier les parties du "discours-code" Forth.

Il existe déjà nombre de classes de mots Forth bien connus des utilisateurs : ce sont les opérateurs de pile, logiques, arithmétiques et de controle du flot d'entrée. Mais ces noms de classes caractérisent le genre d'opérations effectuées par le programme, ils ne décrivent pas des mots qui agissent, des mots sur lesquels on agit, et des mots qui modifient des actions.

On a donc besoin de termes décrivant les interactions entres les mots eux-mêmes. Le terme "phrasé" est utilisé en musique chorale, où il faut respirer entre les phrases. Aucun autre terme musical ne semble convenir, bien que les gens aient besoin de "penser" entres les phrases Forth. Nous trouverons les termes nécessaires en adaptant ceux de la classification standard existant en Grammaire. Une telle adaptation permet de développer un mécanisme créant un Forth "phrasé".

Comment ça marche.

En Forth, toute opération commence et finie avec une pile vide. Entre les deux, des valeurs y sont placées puis retirées. Ainsi une série de mots Forth opérant sur des paramètres étroitement associés doivent être groupés sans espaces supplémentaires. De nouvelles opérations sont normalement précédées d'un espace suplémentaire pour signaler une nouvelle "pensée". Cet espace délimite les positions entre lesquelles une action est complétée.

Les applications Forth sont pathologiques s'il reste quelque chose sur la pile chaque fois qu'elles sont exécutées. Donc l'examen de la pile indique si une tâche est terminée. Mais il apparait que le nombre d'articles ajoutés ou ôtés de la pile est sans rapport avec le "phrasé" en Forth. En effet, la plupart des mots Forth sont construits à partir de plusieurs phrases formées d'autres mots Forth, le mot résultant ayant son propre diagramme de pile. De même le sens des articles de la pile ne joue aucun rôle: ils peuvent être aussi bien des drapeaux sur un bit que des adresses de chaines. L'emplacement des espaces supplémentaires n'a pas grand chose à voir avec le sens des articles empilés.

Termes grammaticaux, leur lien avec les opérations de pile.

Comme en grammaire, un discours Forth est formé de quatre parties: les NOMS,les VERBES,les ADJECTIFS et les INTERJECTIONS.Un NOM est un mot Forth empilant des valeurs: il nomme en effet quelque chose et ouvre un "sujet de discussion" sur la pile. Un autre genre de mot Forth utilise les articles empilés: c'est le VERBE, menant l'action et terminant une phrase en vidant la pile.

Les mots prenant une valeur sur la pile et y déposant cette valeur ou une autre sont des ADJECTIFS: ils modifient les noms et altèrent d'une certaine façon le sujet (les idées empilées) de la discussion. Enfin le dernier genre ignore complètement la pile: les INTERJECTIONS représentent une coupure dans le flot de la conversation et changent momentanément le contexte.

En résumé *(table 1)* :

TABLE 1

GENRE du mot	PILE
NOM	(-- qcq)
ADJECTIF	(qcq -- qcq)
VERBE	(qcq --)
INTERJECTION	(--)

→

Le sens des mots n'intervient que très peu. Les mots DUP, SWAP, OVER, DROP, etc, sont strictement des ADJECTIFS ou des VERBES. Mais le "phrasé" les traite comme des NOMS, car ils se réfèrent à des articles déjà présents dans la discussion (la pile). En d'autres termes, ce sont des PRONOMS puisqu'ils remplacent des noms. Mais pour le "phrasé", leur sens est sans importance.

Construction des phrases.

Dans tous les langages naturels, une phrase est une séquence correcte de mots. En Forth, les mots plaçant des articles sur la pile doivent nécessairement apparaitre avant ceux modifiant ou prenant des valeurs empilées. Comme en français (mais pas en anglais), les adjectifs suivent les noms, alors que les verbes arrivent en dernier (comme en Latin classique).

En langue naturelle, la ponctuation utilise les points et virgules. Pour d'évidentes raisons, on ne peut placer des points entre les phrases Forth. Mais l'insertion d'espaces est sans danger, et d'ailleurs utilisée couramment pour ponctuer les coupures naturelles du code Forth.

Liste des règles.

Nous pouvons maintenant décider dans quels cas des espaces supplémentaires doivent être insérés. Un ensemble raisonnable de règles de "phrasé" serait alors:

Des espaces doivent être ajoutés :

- après un nom s'il est suivi d'une interjection;
- après un adjectif s'il est suivi d'un nom ou d'une interjection;
- après un verbe suivi d'un nom, d'un adjectif ou d'une interjection;
- après une interjection et un nom, ou un adjectif, ou un verbe.

Pas d'espace supplémentaire autrement.

Ces règles sont résumées en *table 2* .

TABLE 2 : ESPACES SUPPLEMENTAIRES.

TYPE	NOM	ADJECTIF	VERBE	INTERJECTION
NOM	-	-	-	+
ADJECTIF	+	-	-	+
VERBE	+	+	-	+
INTERJECTION	+	+	+	-

Le premier mot d'une paire se trouve dans l'une des rangées de la colonne "type". Le second est en tête d'une colonne. A l'intersection, un signe "+" signale qu'un espace suipplémentaire doit être ajouté.

Ceci recouvre toutes les règles du phrasé, sauf pour les mots de structure logique tels que IF, ELSE, et THEN. Ceux-ci ont leur propre convention de formatage, mais il ne peut y avoir de conflit puisque le phrasé concerne les séparations horizontales, alors que le formatage des structures de contrôle est vertical. Formatage et phrasé sont deux façons de rendre le code Forth lisible (cf. BRODIE, o.p.).

Ces mots de structure peuvent être considérés comme des CONJONCTIONS. Remarquez qu'elles n'affectent la pile qu'à la compilation, pas à l'exécution. On pourrait utiliser ce comportement de la pile pour faire de l'impression soignée ("pretty-print") du code Forth, mais il se trouve que cette méthode est moins pratique que celles en usage.

Comment cela fonctionne-t-il en réalité?

Voici des exemples de phrasé basé sur les règles précédentes, avec des définitions possibles de mots familiers (ou peut-être pas tellement ...).

→

```
: >TYPE     ( addr,len -- )    >R   PAD R@ CMOVE    PAD R> TYPE ;

: LIST      ( scr# -- : affiche contenu de l'écran scr#. )
 1 ?ENOUGH   DECIMAL CR ." Scr= "   DUP .   SCR !   L/SCR 0
 DO CR   I 3 .R   SPACE   I C/L *   SCR @ BLOCK +
     C/L -TRAILING >TYPE
 LOOP CR ;

: +         ( n,n -- n : simple addition. )
 BEGIN ?DUP
 WHILE 2DUP XOR   ROT ROT AND 2* REPEAT ;

: T*        ( dn,,n -- tn,,, : triple produit. )
 TUCK >R >R   OVER >R   UM*   0 R> R> R@ OVER >R   0<
 IF D- ELSE 2DROP THEN
 R> R@ OVER >R   UM* D+   R> 0<   R> AND - ;

: T/        ( tn,,,n -- dn,, : triple quotient. )
 DUP >R   ABS   >R   DUP 0<   R@ AND +   R@ UM/MOD
 -ROT R> UM/MOD   NIP SWAP R> ?NEGATE ;

: M*/       ( dn,,n,n -- dn,, : résultat interm. sur 32 oct. )
 >R   T*   R> T/ ;
```

Remarquez que le phrasé est encore convenable, même lorsque des substitutions de mots ont eu lieu. Par exemple, dans la définition de "+", si 2DUP est remplacé par OVER OVER, les règles de base seront encore respectées. On pourrait aussi remplacer les deux ROT par -ROT sans effet néfaste.

(NdT) En Forth 83-Standard :

?ENOUGH	(n --) affiche un message d'erreur si le nombre de paramètres empilés est insuffisant.
TUCK	(n1 n2 -- n2 n1 n2) glisse le premier élément de la pile sous le second.
NIP	efface le deuxième élément de la pile.
L/SCR	nombre de lignes par écran.

Sur quelques points de détail.

L'examen de ces exemples révèle quelques conséquences pratiques de ces règles. Quelques fois l'espace supplémentaire inséré entre les mots doit être un Retour Chariot pour ne pas couper une phrase par une fin de ligne à l'affichage. Mais ceci dépend davantage du "pretty-printing" que du phrasé.

D'autre part, il semble y avoir une classe de mots que l'on pourrait appeler des PREPOSITIONS; ces mots sont liés d'une certaine façon au mot qui les suit. Les mots de définition en sont un bon exemple. VARIABLE, CONSTANT, .", FORGET, CODE et d'autres mots recherchent dans le flot d'entrée un autre mot ou une chaîne comme paramètre; dans le contexte du phrasé Forth par effet de pile, une partie du discours dépend de l'ensemble de cette chaîne. Comme plus d'un mot est toujours nécessaire, peut-être devrait-on les appeler des PHRASES PREPOSITIONNELLES. Une telle phrase peut être constituée de n'importe lequel des quatre types de mots définis. Par conséquent, les PREPOSITIONS elles-mêmes sont différentes et demandent un traitement spécial.

En dehors de ces exceptions, on peut considérer que la plupart des mots sont des adjectifs. Cette "règle" est assez sûre car une analyse des mots Forth montre que les ADJECTIFS sont deux fois plus nombreux que toute autre espèce de mot. De plus, si on étend la table 2 pour y inclure des mots INCONNUS, l'arrangement des "+" et des "-" pour ces colonnes et rangées INCONNUS se trouvent être identiques à ceux des ADJECTIFS: le phrasé n'y voyant aucune différence, ces mots INCONNUS peuvent donc être traités comme des ADJECTIFS.

Phrasé automatique.

Nous possédons désormais un ensemble raisonnable et complet de règles applicables au phrasé. Nous en savons assez pour construire un algorithme, nous épargnant peut-être beaucoup de frustration en édition d'écran.

L'implantation d'un "phraseur" automatique peut se faire de différentes façons. On peut par exemple créer un nouveau vocabulaire (nommé PRETTY-PHRASER) contenant tous les noms de mots explicitement déclarés comme NOM, ADJECTIF, VERBE ou INTERJECTION. Une autre approche serait d'ajouter une petite structure à la définition des mots du dictionnaire. Ceci peut être fait facilement, trois bits au plus sont nécessaires pour faire la sélection à partir des

→

cas connus. Dans le modèle F83 de Laxen/Perry, l'information peut être placée dans quelques bits inutilisés du champ VIEW, à moins que vous ayez plus de 512 blocs dans vos fichiers source.
(NdT: le champ VIEW donne le numéro d'écran contenant la définition du mot qui le suit, ainsi que sa syntaxe...)

Dans tous les cas, pendant le "Pretty-Printing", cette information est accessible et des espaces peuvent être insérés convenablement selon les règles énoncées ci-dessus.

Plutôt que d'entrer les genres des mots à la main, on peut construire un analyseur grammatical simple examinant les commentaires des diagrammes de pile dans chaque définition. Cet analyseur est appelé soit lorsque le PRETTY-PHRASER est actif, soit pendant la compilation du mot dans le dictionnaire, cela dépendant de l'implantation du "phrasé". Comme le nombre d'articles sur la pile n'intervient pas, la présence ou l'absence de caractères de chaque côté des "--" donne toutes les informations nécessaires pour convertir la table 1 directement en code Forth.

Il y aura toujours des exceptions, mais aucune ne sera sans remède. Pour ceux n'ayant pas les diagrammes de pile dans les écrans-source, le cas par défaut serait l'ADJECTIF pour les raisons déjà exposées. VARIABLE et CONSTANT peuvent être traitées soit par du code ajouté aux définitions elles-mêmes, soit spécialement dans le PRETTY-PHRASER lorsqu'elles se présentent.

<u>Quelques applications.</u>

Ayant été réduit à un petit nombre de règles, et étant automatisé, le phrasé en Forth n'est plus un "art subjectif". L'approche donnée ici peut devenir un article de plus dans la liste des utilitaires modernes de Forth, avec des retombées insoupçonnées.

En plus des listings, le phrasé automatique peut nettoyer des écrans surchargés par de multiples éditions et devenus illisibles. Cela demande du soin car certains blocs vont "pretty-phraser" plus de 16 lignes. Cependant, si seule l'impression sur papier est désirée, les blocs Forth peuvent être comprimés au maximum tant qu'un "pretty-phraser" est disponible pour "décomprimer" le source.

Si la documentation accompagnant un programme Forth est de la plus haute importance, l'emplacement des espaces supplémentaires peut être utilisé pour diriger les programmeurs les plus scrupuleux vers des endroits où des commentaires peuvent être placés.

Et pour ceux faisant un usage intensif du décompilateur Forth, la sortie d'un tel décompilateur peut être utilisée pour exposer le travail interne du système. Quel autre langage opérant sur le système peut en faire autant ? ■

Une série de petites routines nous a été aimablement communiquée par un de nos adhérents, Mr G. SOULA. Ces routines sont applicables à la majorité des systèmes existants. Certaines, bien que diffusées sans aucun commentaire, ne devraient poser guère de difficulté de compréhension.

<u>Références:</u>
Difficulté de programmation: moyenne
Catégorie: utilitaire
Difficulté d'exercice: moyenne

<u>L'exercice:</u>
L'exécution vectorisée permet de faire faire à un même mot différentes choses.

<u>Le programme:</u>
Le principe de la vectorisation est qu'au lieu d'invoquer un mot pour le faire exécuter, on fait <u>EXECUTE</u> sur le cfa de ce mot, ce cfa pouvant être contenu <u>dans une</u> variable.

Le mot <u>@EXECUTE</u> lit le contenu de la variable de vectorisation et effectue la vectorisation à condition que le cfa ne soit 0.

L'exemple classique peut se schématiser ainsi: à partir d'une variable <u>ACTION</u> on peut faire afficher plusieurs textes différents:

NOW MR ETAT-CIVIL affiche Monsieur
NOW ME ETAT-CIVIL affiche Madame etc...

Une application plus intéressante est un générateur de règles où <u>RULE</u> est la variable de vectorisation:

<u>?REGLES</u> demande si on désire voir les règles du jeu,
si oui, les règles sont affichées,
si non, on passe au jeu.

<u>!RULE</u> transforme le pfa en nfa et le stocke dans <u>RULE</u>.

Le mot <u>CASE:</u>; les structures de case constituent une extension des variables de vectorisation par la création d'un tableau vectorisé.

<u>CASE:</u> est un mot de définition qui va:
- créer un nouveau mot et stocker le cfa des mots qui suivent jusqu'au mot <u>;</u>.
- à l'exécution, il y a vectorisation sur le numéro de la structure de case.

Exemple: 0 ETAT-CIVIL affiche Monsieur
1 ETAT-CIVIL affiche Madame etc...

```
 0 ( EXECUTION VECTORISEE )
 1
 2 : @EXECUTE   ( ad - )   @ ?DUP
 3    IF EXECUTE THEN ;
 4
 5
 6 VARIABLE ACTION
 7
 8 : MR ." Monsieur " ;
 9 : ME ." Madame " ;
10 : ML ." Mademoiselle " ;
11 : EN ." Enfant " ;
12
13 : ETAT-CIVIL   ACTION @EXECUTE ;
14 : NOW  [COMPILE]  ' CFA ACTION ! ;
15
```

```
 0 ( EXECUTION VECTORISEE )
 1
 2 VARIABLE RULE
 3
 4 : ?REGLES ." Voulez vous voir les regles O/N?" O/N
 5    IF RULE @EXECUTE THEN ;
 6
 7 : !RULE   ( pfa - )   CFA RULE ! ;
 8
```

Suite page 14

REALISATION D'UN COMPILATEUR BLAISE

Ce compilateur sera réalisé en utilisant l'analyseur précédemment dévellopé comme squelette et en insérant les instructions de génération de code objet. celui-ci sera du FORTH compilé.

On utilise alors deux variables supplémentaires:

ADR qui contient le pfa du dernier identificateur de procédure lu, ou le numéro de la dernière variable.

NUM qui contient la valeur du dernier litéral ou de la dernière constante ou bien le niveau de la dernière variable.

Le seul point appelant des commentaires est la gestion des niveaux des procédures, avec les variables locales et la possibilité de récursivité.

Les variables sont stockées sur la pile, pour permettre une gestion dynamique (les variables existent seulement pendant l'activation de la procédure correspondante: il faut donc bien les distinguer du mot déclaré par DEFVAR qui ne sert que durant la compilation), avec une structure de liste chaînée. On a en fait deux chaînes de pointeurs: les pointeurs dynamiques (DL= Dynamic Link), qui rendent compte de l'ordre chronologique des allocations, et les pointeurs statiques (SL) qui rendent compte des niveaux de procédure. prenons un exemple: soit le programme

```
Procédure A ;
  Procédure B ;
    Procédure C ;
      Begin ( C )
      ... B ( récursif ) ...
      End ( C )
    Begin ( B )
    ... Call C ...
    End ( B ).
  Begin ( A )
  ... Call B ...
  End ( A )
```

A est de niveau 1, B de niveau 2 et C est de niveau 3. Après l'appel récursif de B par C, la structure de la pile est la suivante:

```
O SLA
O DLA
variables A
SLB
DLB
variables B
SLC
DLC
variables C
SLB'          Lastbase
DLB'
variables B
```

Une adresse de variables contient donc la différence de niveau (pour exemple, 1 pour une variable de B appelée par C) et le numéro de cette variable dans son bloc. La procédure BASE permet de remonter la chaîne des SL.

Il est possible maintenant d'exécuter un petit programme Blaise: les tours de HANOI. Pour ceux qui ne connaissent pas ce grand classique, en voici le principe: il faut déplacer n anneaux de diamètres décroissants d'un piquet sur un autre en utilisant un troisième piquet. On ne peut déplacer qu'un anneau à la fois, et ne le poser que sur un anneau de plus grand diamètre.

Après avoir rentré le programme, il suffit de faire 1 PROGRAMME HANOI pour le compiler, puis après le message "COMPILATION TERMINEE", faire HANOI n pour le lancer. Les anneaux sont initialement sur le piquet numéro 1, et à la fin sur le numéro 2. Les couples affichés donnent le piquet de départ et le piquet d'arrivée de l'anneau à déplacer.

Le BLAISE n'accepte pas d'argument pour les procédures mais l'utilisation de variables globales permet de pallier ce défaut, comme vous le voyez dans HANOI.

Fin (pour l'instant ....)
Eric AUBOURG

```
SCR # 18
  Ø ( *** COMPILATEUR BLAISE 1 *** )
  1
  2 VOCABULARY MOTRESERVES MOTRESERVES DEFINITIONS
  3  1 CONSTANT BEGIN     2 CONSTANT CALL
  4  3 CONSTANT CONST     4 CONSTANT DO
  5  5 CONSTANT END       6 CONSTANT IF
  6  7 CONSTANT ODD       8 CONSTANT PROCEDURE
  7  9 CONSTANT THEN     1Ø CONSTANT VAR
  8 11 CONSTANT WHILE    12 CONSTANT READ
  9 13 CONSTANT WRITE    15 CONSTANT )
 1Ø 17 CONSTANT :=       18 CONSTANT +
 11 19 CONSTANT -        2Ø CONSTANT *
 12 21 CONSTANT /        22 CONSTANT =
 13 23 CONSTANT >        24 CONSTANT <
 14 25 CONSTANT <>       26 CONSTANT <=
 15 27 CONSTANT >=       28 CONSTANT ;          -->

SCR # 19
  Ø ( *** COMPILATEUR BLAISE 2 *** )
  1 14 CONSTANT (        16 CONSTANT ,
  2 29 CONSTANT .        3Ø CONSTANT -->
  3 FORTH DEFINITIONS
  4 Ø VARIABLE ADR       Ø VARIABLE NUM
  5 ' MOTRESERVES CONSTANT LIMITVOC
  6 : ERREUR
  7   ." ERREUR #" . BLK @ ." BLOC #" . IN @ 64 / ." LIGNE #"
  8   . CR HERE COUNT TYPE QUIT ;
  9 : GETSYM
 1Ø   DROP -FIND
 11   IF DROP CFA DUP LIMITVOC > ELSE Ø  DUP THEN
 12   IF EXECUTE DUP 3Ø =
 13     IF [COMPILE] --> [ SMUDGE ] GETSYM [ SMUDGE ] THEN
 14
 15         -->
```

→

```
SCR # 20
  0 ( *** COMPILATEUR BLAISE 3 *** )
  1  ELSE
  2  DROP HERE NUMBER DROP NUM ! 31
  3  THEN ;
  4 : DEFCONST ( LEVEL SYM --- LEVEL PFA SYM )
  5    <BUILDS HERE SWAP 0 ,
  6    DOES> @ NUM ! 33 ; ( VALEUR -> NUM, --- 33 )
  7 : DEFVAR   ( LEVEL # SYM )
  8    <BUILDS >R OVER OVER , , R>        ( EXEC : # -> ADR,      )
  9    DOES> DUP @ ADR ! 2+ @ NUM ! 32 ; ( LEVEL -> NUM, --- 32 )
 10 : DEFPROC ( LEVEL SYM -- PFA LEVEL SYM )
 11    <BUILDS HERE ROT ROT 0 , OVER , 0 ,
 12    DOES> ADR ! 34 ; ( PFA -> ADR, --- 34 )
 13 0 VARIABLE LASTBASE
 14 CREATE SPSTO ( N --- , N->SP )
 15 HEX F9E1 , E9FD , DECIMAL  SMUDGE       -->

SCR # 21
  0 ( *** COMPILATEUR BLAISE 4 *** )
  1 : BASE ( LEVEL APPELANT-APPELE --- ADRESSE )
  2    LASTBASE @ SWAP DUP 0 >
  3    IF 0 DO @ LOOP
  4    ELSE DROP THEN ;
  5 : APPEL ( ARGUMENT COMPILE PFA , LEVEL APPELANT )
  6    R> DUP 4 + >R DUP 2+ @ SWAP @ ( LEVEL PFA )
  7    DUP >R 2+ @ - BASE LASTBASE @ 2 - ( POINTEURS SL DL )
  8    SP@ DUP 2+ LASTBASE ! R 4 + @ - SPSTO
  9    R> @ EXECUTE
 10    LASTBASE @ 2 - SPSTO 2+ LASTBASE ! DROP ;
 11 : (RCLVAR) ( ADR DLEV --- VALEUR )
 12    BASE SWAP - @ ;
 13 : RCLVAR ( LEVEL SYM )
 14    COMPILE LIT ADR @ , COMPILE LIT OVER NUM @ - ,
 15    COMPILE (RCLVAR) ;    -->


SCR # 22
  0 ( *** COMPILATEUR BLAISE 5 *** )
  1 : STOVAR   ( VAL ADR LEV --- )
  2    BASE SWAP - ! ;
  3 : INPUT
  4    BEGIN
  5     BL WORD HERE 1+ C@ 0=
  6    WHILE
  7     QUERY
  8    REPEAT
  9    HERE NUMBER DROP ;
 10 0
 11 : FACTEUR     ( LEVEL SYM )
 12    DUP 31 = OVER 33 = OR
 13    IF COMPILE LIT NUM @ , GETSYM  ELSE
 14    DUP 32 = IF RCLVAR GETSYM ELSE
 15    DUP 34 = IF 21 ERREUR ELSE      -->


SCR # 23
  0 ( *** COMPILATEUR BLAISE 6 *** )
  1    DUP 14 = IF GETSYM [ HERE SP@ 18 + ! 0 , ]
  2     15 - IF 22 ERREUR THEN DUP GETSYM
  3    ELSE 23 ERREUR
  4    THEN THEN THEN THEN  ;
  5 : TERME ( LEVEL SYM )
  6    FACTEUR
  7    BEGIN
  8     DUP 20 = IF 1 >R R ELSE
  9     DUP 21 = IF 0 >R 1 ELSE
 10     0 THEN THEN
 11    WHILE
 12     GETSYM FACTEUR R>
 13     IF     COMPILE *
 14     ELSE   COMPILE / THEN
 15    REPEAT  ;    -->
```

```
SCR # 24
  Ø ( *** COMPILATEUR BLAISE 7 *** )
  1 : EXPRESSION    ( LEVEL SYM )
  2    DUP 18 = IF GETSYM Ø ELSE
  3    DUP 19 = IF GETSYM 1 ELSE
  4    Ø THEN THEN >R
  5    TERME
  6    R> IF COMPILE MINUS THEN
  7    BEGIN
  8      DUP 18 = IF 1 >R I ELSE
  9      DUP 19 = IF Ø >R 1 ELSE
 1Ø      Ø THEN THEN
 11    WHILE
 12      GETSYM TERME R>
 13      IF COMPILE +
 14      ELSE COMPILE - THEN
 15    REPEAT ;             ' EXPRESSION CFA SWAP !         -->

SCR # 25
  Ø ( *** COMPILATEUR BLAISE 8 *** )
  1 : CONDITION    ( LEVEL SYM )
  2    DUP 7 =
  3    IF GETSYM EXPRESSION COMPILE 2 COMPILE MOD
  4    ELSE EXPRESSION
  5      DUP 22 < OVER 27 > OR
  6      IF 2Ø ERREUR THEN  >R
  7      DUP GETSYM EXPRESSION R>
  8      CASE
  9        22 OF COMPILE = ENDOF
 1Ø        23 OF COMPILE > ENDOF
 11        24 OF COMPILE < ENDOF
 12        25 OF COMPILE = COMPILE Ø= ENDOF
 13        26 OF COMPILE > COMPILE Ø= ENDOF
 14        27 OF COMPILE < COMPILE Ø= ENDOF
 15      ENDCASE THEN ;       -->

SCR # 26
  Ø ( *** COMPILATEUR BLAISE 9 *** )
  1 : INSTRUCTION     ( LEVEL SYM )
  2    [ SMUDGE ] DUP
  3    CASE
  4      32 OF ADR @ NUM @ >R >R GETSYM 17 -
  5            IF 13 ERREUR THEN
  6            DUP GETSYM EXPRESSION COMPILE LIT R> ,
  7            COMPILE LIT OVER R> - , COMPILE STOVAR ENDOF
  8      33 OF 12 ERREUR ENDOF
  9      34 OF 12 ERREUR ENDOF
 1Ø       2 OF GETSYM DUP 34 =
 11            IF COMPILE APPEL ADR @ , OVER , GETSYM
 12            ELSE 14 ERREUR THEN ENDOF
 13       6 OF GETSYM CONDITION [COMPILE] IF >R >R DUP 9 =
 14            IF GETSYM INSTRUCTION R> R> [COMPILE] ENDIF
 15            ELSE 16 ERREUR THEN ENDOF  -->

SCR # 27
  Ø ( *** COMPILATEUR BLAISE 1Ø *** )
  1     11 OF [COMPILE] BEGIN >R >R GETSYM CONDITION DUP 4 = IF
  2           GETSYM [COMPILE] WHILE >R >R INSTRUCTION R> R> R> ROT
  3           ROT R> ROT ROT [COMPILE] REPEAT
  4           ELSE 18 ERREUR THEN ENDOF
  5      1 OF GETSYM BEGIN INSTRUCTION DUP 28 = WHILE GETSYM
  6     REPEAT DUP 5 - IF 17 ERREUR THEN GETSYM ENDOF
  7    ENDCASE
  8    DUP 12 = IF 1 1 ELSE DUP 13 = IF Ø 1 ELSE Ø THEN THEN
  9    IF >R GETSYM DUP 14 - IF 25 ERREUR THEN
 1Ø      GETSYM
 11      BEGIN DUP 32 - IF 42 ERREUR THEN
 12            I IF COMPILE INPUT COMPILE LIT ADR @ ,
 13                   COMPILE LIT OVER NUM @ - , COMPILE STOVAR
 14               ELSE RCLVAR COMPILE . THEN
 15            GETSYM DUP 16 =       -->
```

```
SCR # 28
  Ø ( *** COMPILATEUR BLAISE 11 *** )
  1          WHILE GETSYM
  2          REPEAT R> DROP
  3          DUP 15 - IF 22 ERREUR THEN
  4          GETSYM COMPILE CR THEN ;  SMUDGE
  5 : BLOC     ( LEVEL SYM )
  6    SWAP 1+ SWAP    [ SMUDGE ]
  7    DUP 3 =    ( CONSTANTE )
  8    IF BEGIN
  9         DEFCONST    ( LEVEL PFA SYM )
 1Ø         GETSYM DUP 22 - IF 3 ERREUR THEN
 11         GETSYM 31 - IF 2 ERREUR THEN
 12         NUM @ SWAP ! Ø GETSYM DUP 16 -
 13       UNTIL
 14       DUP 28 - IF 5 ERREUR THEN GETSYM
 15    THEN  2 >R          -->

SCR # 29
  Ø ( *** COMPILATEUR BLAISE 12 *** )
  1    DUP 1Ø =      ( VARIABLE )
  2    IF R> SWAP  ( LEVEL # SYM )
  3       BEGIN
  4         SWAP 2+ SWAP DEFVAR GETSYM DUP 16 -
  5       UNTIL SWAP >R   ( # VAR SUR RS )
  6       DUP 28 - IF 5 ERREUR THEN GETSYM
  7    THEN
  8    BEGIN
  9      DUP 8 =    ( PROCEDURE )
 1Ø    WHILE
 11      DEFPROC    ( PFA LEVEL SYM )
 12      GETSYM 28 - IF 5 ERREUR THEN
 13      LATEST >R Ø GETSYM BLOC R> CURRENT @ !
 14      DUP 28 - IF 5 ERREUR THEN GETSYM
 15    REPEAT      -->

SCR # 3Ø
  Ø ( *** COMPILATEUR BLAISE 13 *** )
  1    ROT HERE OVER ! 4 + R> 2 - SWAP !
  2    [ ' : 18 + ] LITERAL , [COMPILE] ]
  3    INSTRUCTION COMPILE ;S
  4    SWAP 1 - SWAP ; SMUDGE
  5 : INIT Ø  SP@ LASTBASE ! Ø R> DUP 2+ >R @ >R SP@ R 4 + @ - SPSTO
  6    R> @ EXECUTE CR ." EXECUTION TERMINEE" SP! ;
  7 : PROGRAMME   ( #BLOC ---      PROGRAMME NOM )
  8    HERE Ø , Ø , Ø , [COMPILE] : COMPILE INIT DUP ,
  9    [COMPILE] ; SWAP
 1Ø    BLK @ >R IN @ >R B/SCR * BLK ! Ø IN !
 11    MOTRESERVES DEFINITIONS CR [COMPILE] ]
 12    Ø DUP GETSYM BLOC 29 - IF 9 ERREUR THEN
 13    DROP [COMPILE] FORTH DEFINITIONS [COMPILE] [
 14    R> IN ! R> BLK ! ." COMPILATION TERMINEE "
 15 ;
```

```
SCR # 1
  Ø VAR N , I , J ;
  1 PROCEDURE DEPLACER ;
  2 VAR N1 , I1 , J1 ;
  3 BEGIN
  4   IF N = 1 THEN WRITE ( I , J ) ;
  5   IF N > 1 THEN
  6   BEGIN
  7     I1 := I ; J1 := J ; N1 := N ;
  8     N := N1 - 1 ; J := 6 - I - J ;
  9     CALL DEPLACER ;
 1Ø     N := 1 ; J := J1 ; I := I1 ;
 11     CALL DEPLACER ;
 12     N := N1 - 1 ; I := 6 - I1 - J1 ; J := J1 ;
 13     CALL DEPLACER
 14   END
 15 END ;  -->

SCR # 2
  Ø BEGIN
  1   READ ( N ) ; I := 1 ; J := 2 ;
  2   CALL DEPLACER
  3 END .
  4
  5
```

 par Y. BESOMBES

```
     6
0 ( routines machine pour echanges memoires extra segment )
1 ( long-store,long-peek,log-move )                        5
2  code l! ds ax mov dx pop dx ds mov bx pop 0 'bx§
3    pop ax ds mov next c; ( val adr2 seg2 ---- )
4  code lc! ds cx mov dx pop dx ds mov
5    bx pop ax pop al 0 'bx§ mov  cx ds mov next c; ( idem )
6       hex f000 constant video decimal
7  code là ds ax mov dx pop dx ds mov    ( adr2 seg2 --- val )
8    bx pop 0 'bx§ push ax ds mov next c;
9  code lcà ds cx mov dx pop dx ds mov    ( idem )
10   bx pop ax ax sub 0 'bx§ al mov ax push cx ds mov next c;
11 : videoà video là 64 - ;
12 : blkà scr à block ;            ( --- )
13
14 10 load 11 load
15 -->
```

```
      7
 0  : a-scr scr ! put_e ; : esc 27 emit ;
 1  : savcur esc 106 emit ; : setcur esc 107 emit ;
 2  : m0 ."          " ;  : mi ." INSERT " ;
 3  : boucle 13 5  at begin  ( boucle editeur pleine page )
 4  key dup case 131 of esc 68 emit endof  ( ar )
 5       13  of esc 76 emit endof
 6       132 of esc 66 emit endof  ( up )
 7       127 of esc 78 emit endof
 8       133 of esc 67 emit endof
 9       144 of savcur 70 1 at m0 setcur esc 79 emit endof
10       134 of esc 65 emit endof
11       18  of savcur 70 1 at mi setcur esc 64 emit endof
12       146 of esc 77 emit endof endcase
13       dup 127 < if dup emit then 149 = until ;
14 : cadre 12 4 at 66 1 do 196 emit loop 12 21 at 66 1 do 196 emit
15   loop ; : edwind 77 14 21 6 window ;   -->
```

```
      8
 0 ( editeur pleine page avec acces direct memoire ecran Apricot )
 1    variable ino : m5 1 1 at ." no  : " ;
 2 : m1 1 1 at ." Ecran no : "  . ;  : m2 1 23 at ." copie" ;
 3 : ecr cls m1 cadre edwind a-scr boucle m2 get-e esc 79 emit ;
 4 : put update flush ; : getno cls m5 query interpret ;
 5 : m3 1 1 at ."     E)dit           Q)uit        W)rite " ;
 6 : edit cls
 7       begin m3 key dup 101 = if getno dup ecr endwind cls then
 8                   dup 119 = if put then
 9                       113 = until cls ;
10
11
12 ( fin editeur  )
13
14
15
```

```
     9
( fonction case f83 )
: case csp à !csp  ; immediate
: of compile over compile  = compile ?branch  >mark
  compile drop ; immediate
: endof compile branch  >mark  swap
 >resolve ; immediate
: endcase compile drop begin spà csp à <> while
  >resolve  repeat csp ! ; immediate
: jour case cls 1 of ." Lundi" endof 2 of ." Mardi" endof
           3 of ." Mercredi" endof 4 of ." Jeudi" endof
           5 of ." Vendredi" endof  6 of ." Samedi" endof
           7 of ." Dimanche" endof endcase  ;
```

```
    10
( routine affichage et lecture d'une chaine sur ecran )
( chaine definie par adr lgr,ligne et col:positions sur ecran )
   ( ADR LGR LIGNE COL ------ )
 : adrr 1- 2 * swap 1- 160 * + ; ( lign col---offset )
 : WRITE adrr SWAP  2* rot swap 0 DO 2DUP
   I 2 / + Cà 64 + SWAP  VIDEO SWAP I + swap L! 2 +LOOP 2DROP ;
( adr lgr ligne col ------ )
 : get  adrr swap 2*  0 do 2dup
   i + videoà  swap i 2 / + c! 2 +loop 2drop ;
```

```
    11
( ecriture lecture ecran video d'un block forth 1024 octets )
: put_e  22 6 do i 6 - 64 * blkà + 64 i 14 write loop ;
: get-e  22 6 do i 6 - 64 * blkà + 64 i 14 get loop ;
```

écran 6: gestion extra segment
écran 9: case-of-endof-endcase
écran 7-8 et 10-11: éditeur pleine page
pour APRICOT PC (et SIRIUS peut-être...)
écran 28-32: routines graphiques GSX sous MSDOS

```
      12
0
1 ( ADR LGR LIGNE COL ------ )  : AF ROT SWAP ;
2  : adrr 1- 2 * swap 1- 160 * + ; ( lign col---adress )
3  : WRITE adrr SWAP  2* AF 0 DO 2DUP
4    I 2 / + Cà 64 + SWAP  VIDEO SWAP I + swap L! 2 +LOOP 2DROP ;
5 ( adr lgr ligne col ------ )
6  : get  adrr swap 2*  0 do 2dup
7    i + videoà  swap i 2 / + c! 2 +loop 2drop ;
8
```

```
      13
0 ( long-move lcmove:dataseg->a2:s2 ;lcmove>:a2:s2->dataseg )
1 variable tp 100 allot
2
3 ( adr1 noctets adr2 seg2 -------)
4 : lcmove rot 0 do 3dup rot i + cà rot i + rot lc! loop  ;
5 : lcmove> rot 0 do 3dup swap i + swap lcà swap i + c! loop ;
6
```

```
      28
0 ( EXTENSIONS GRAPHIQUES GSX)                                 5
1   VARIABLE CONTRL 20 ALLOT   VARIABLE INTIN 160 ALLOT
2   VARIABLE TABLE 200 ALLOT   VARIABLE PTSIN 200 ALLOT
3   code intgsx cx pop dx pop 224 int next c;
4   VARIABLE PTSOUT 200 ALLOT   VARIABLE INTOUT 90 ALLOT
5 : ADR PTSOUT INTOUT PTSIN INTIN CONTRL ;
6 : int224 table 1139 intgsx ;
7 : GSX  20 2 DO DSEG TABLE  I + ! 4 +LOOP
8   ADR 20 0 DO TABLE I +  ! 4 +LOOP  INT224 ;
9 : CONTRL1      CONTRL  ! ;
10 : CONTRL2      CONTRL 2+  ! ;
11 : CONTRL4      CONTRL 6 +  ! ;
12 : INTIN1      INTIN  ! ;
13 : PTSIN1      PTSIN  ! ;  : PTSIN2 PTSIN 2+  ! ;
14 : PTSIN4  PTSIN 6 + ! ; : PTSIN5  PTSIN 8 + ! ;
15 --)
```

```
      29
0 0 CONSTANT CMD  2 CONSTANT CL  : INTIN2 INTIN 2+ ! ;
1 0 CONSTANT ATTRIBUT  : INTIN3 INTIN 4 + ! ;
2 3 CONSTANT CLR  : INTIN4 INTIN 6 + ! ; : INTIN5 INTIN 8 + ! ;
3 6 CONSTANT PLINE  : INTIN6 INTIN 10 + ! ;
4 8 CONSTANT TEXT     : PTSIN3 PTSIN 4 + ! ;
5 : PTSIN7 PTSIN 12 + ! ; : PTSIN6 PTSIN 10 + ! ;
6 : PTSIN8 PTSIN 14 + ! ; : INTIN7 INTIN 12 + ! ;
7 : PTSIN9 PTSIN 16 + ! ; : INTIN8 INTIN 14 + ! ;
8 : PTSIN10 PTSIN 18 + ! ; : INTIN9 INTIN 16 + ! ;
9 : INTIN10 INTIN 18 + ! ; : CONTRL3 CONTRL 4 + ! ;
10 : CLEAR CLR CONTRL1 0 CONTRL2 GSX ;
11 : EXT CL CONTRL1 0 CONTRL2 GSX  ;
12 : CONTRL6 CONTRL 10 + ! ;
13 : INIT CLEAR 1 CONTRL1 0 CONTRL2 10 CONTRL4 1 INTIN1 1 INTIN2
14   1 INTIN3 1 INTIN4 1 INTIN5 1 INTIN6 1 INTIN7 1 INTIN8
15   1 INTIN9 1 INTIN10 GSX ; --)
```

```
0
." coucou"  cr
." c'est moi"
```

```
0
." coucou"  cr
." c'est moi"
```

```
      30
0
1
2
3
4
5 : LINETYPE INTIN1 15 CONTRL1 0 CONTRL2 GSX ;
6 : LINECOLOR INTIN1 17 CONTRL1 0 CONTRL2 GSX ;
7 : LINEMARKER INTIN1 18 CONTRL1 0 CONTRL2 GSX ;
8
9  : INITAB CONTRL 20 0 FILL ." OK" INTIN 160 0 FILL ." OK"
10    TABLE 200 0 FILL ." OK" PTSIN 200 0 FILL ." OK" ;
11
12 : PT 20 0 DO PTSIN I + ? 2 +LOOP ;
13 : IN 20 0 DO INTIN I + ? 2 +LOOP ;
14 : CTL 20 0 DO CONTRL  I + ? 2 +LOOP ;
15 --)
```

```
   31
: LINE DUP CONTRL2 4 * 0 DO PTSIN I + ! 2 +LOOP 6 CONTRL1
GSX ;  : DOT 2DUP 2 LINE ;
: CIRCLE PTSIN5 PTSIN2 PTSIN1 11 CONTRL1 3 CONTRL2 4 CONTRL6
  GSX ;
: MARKER DUP CONTRL2 4 * 0 DO PTSIN I + ! 2 +LOOP 7 CONTRL1
  GSX ;
( attribut des polymarkers )
: ATTRIB INTIN1 CONTRL1 0 CONTRL2 GSX ;
: HOLLOW 23 0 ATTRIB ; : SOLID 23 1 ATTRIB ; : HATCH 23 3
  ATTRIB ; : VERT 24 1 ATTRIB ; : HOR 24 2 ATTRIB ;
: DOTC 18 1 ATTRIB ; : ASTERIX 18 2 ATTRIB ; : CIRC 18 4
  ATTRIB ;
: SOL 15 1 ATTRIB ; : DASH 15 2 ATTRIB ;
: LDOT 15 3 ATTRIB ;

cr .( **GSX**) cr
```

```
   32
: EX2 INIT HOR HATCH 20 1 DO 1000 I * 1000 I * 360 I * CIRCLE
2 +LOOP SOLID  20 1 DO 1000 I * 32000 1000 I * - 360
    I * CIRCLE 2 +LOOP HATCH VERT 20 1 DO 32000 1000 I * -
1000 I * 360 I * CIRCLE 2 +LOOP HOLLOW 20 1 DO 32000 1000 I *
- 32000 1000 I * - 360 I * CIRCLE 2 +LOOP KEY EXT ;
```

Dans toute science, le besoin de faire référence à un fond de connaissances commun est resté une préoccupation majeure. La science informatique n'échappe pas non plus à cette nécessité. Mais plus que partout ailleurs, dans un domaine où la progession entre les générations est mesurée en années, voire en mois, et non en décénnies, imposer un standard reste une gageure.

C'est ce pari insensé que relèvent deux américains, Laxen et Perry, en proposant un système de développement complet à l'ensemble de la communauté FORTH internationale. Ce système, au standard 83, est diffusé dans le domaine public, ce qui est à notre avis, la meilleure garantie contre toute reproduction illicite, les auteurs poussant la délicatesse à encourager la reproduction sous toutes les formes et par tous les moyens de leur oeuvre.

LE FORTH 83-STANDARD SOUS CP/M

Cette version, disponible sur tous les systèmes tournant sous CP/M 2.2 et MSDOS, est diffusée généralement sous forme de disquette, dont le contenu sous CP/M est le suivant:

```
F83      .COM      KERNEL    .HEX
META80   .BQK      EXTEND80  .BQK
UTILITY  .BQK      DIRECT    .BQK
README   .TXT      USQ       .COM
```

Le premier fichier, 'F83.COM', est le programme FORTH principal. Il contient sous forme compilée le contenu des fichiers 'UTILITY', 'EXTEND80' et 'META80'. Sous MSDOS, le fichier 'F83.COM' est nommé 'RUNME.COM'.

LA DECOMPRESSION DE FICHIERS SOUS CP/M

Les fichiers d'extension 'BQK' contiennent les programmes source, sous forme compresssée, à partir desquels a été réalisée la génération du programme 'F83'. Pour pouvoir lire et compiler leur contenu, il est donc nécessaire de les décompresser. Pour cela, taper sous CP/M:

```
USQ nomfichier.ext
```

Le fichier à décompresser doit être un fichier de type compressé. Pour exemple, la commande 'USQ F83.COM' n'est pas valide.

Un fichier décompressé par 'USQ' prend l'extension 'fichier.BLK'. Son contenu peut dorénavant être lu sous FORTH en tapant 'OPEN fichier.BLK' où 'fichier' correspond au nom des fichiers qui ont été décompressés. Exemple:

```
OPEN DIRECT.BLK
```

puis la lecture de son contenu par 'n LIST', où n est un numéro de bloc.

LA META-GENERATION

Le système 83-Standard a été généré de manière tout à fait révolutionnaire: il a été 'méta-compilé' et non assemblé. Cette méthode constitue en effet une nouveauté dans le monde de la micro-informatique, car réservée jusqu'alors aux gros systèmes. La méta-compilation, en FORTH, consiste à créer un noyau de base à partir d'un programme source traité par FORTH lui-même. A partir de ce noyau, une nouvelle version peut être compilée. Mais on peut aussi compiler une application spécifique. Pour peu que le programme en question soit standard, il pourra s'exécuter sur n'importe quel système, ce qui est le cas des programmes sources écrits en 83-Standard, lesquels tournent, pour exemple, aussi bien sur AMSTRAD que sur IBM et compatibles.

Ce souci de la portabilité a été poussé à l'extrême avec le programme 'F83.COM' qui a pu être transféré sur à peu près tous les systèmes tournant sous CP/M 2.2, y compris des systèmes aussi exotiques que le BONDWELL 2 ou la série AMSTRAD /SCHNEIDER CPC et PCW. De nombreuses définitions, vectorisées avec le système profond, peuvent voir leur comportement modifié de manière très simple:

```
: nouveau-vecteur
   ... définition ... ;
' nouveau-vecteur IS vecteur
```

Ces altérations peuvent porter sur à peu près l'ensemble du système de gestion des entrées sorties.

LE NOYAU FORTH

Le fichier 'KERNEL.HEX' résulte de la seule méta-compilation du contenu du fichier 'META80.BLK'. Ce fichier est fourni au format '.HEX', ce qui permet son transfert entre deux systèmes par modem ou RS232 à l'aide de la seule commande PIP sous CP/M.

Pour pouvoir être exécutable, il doit être converti en fichier '.COM' en tapant la commande 'LOAD KERNEL.HEX' sous CP/M. Maintenant vous pouvez démarrer 'KERNEL', ce qui vous met sous FORTH.

L'exécution d'une nouvelle méta-compilation se fait en spécifiant à la suite de KERNEL le fichier d'origine de votre version FORTH personnalisée. Exemple:

```
KERNEL EXTEND80.BLK      sous CP/M, puis
START                    sous FORTH.
```

Une nouvelle génération de FORTH, telle qu'elle est définie précédemment fait appel successivement au contenu des fichiers 'EXTEND80.BLK', 'CPU8080.BLK' et 'UTILITY.BLK'. La méta-compilation ne pourra s'exécuter correctement que si tous ces fichiers figurent sur le même disque actif. Si la capacité des disquettes de votre système est insuffisante, il faudra apporter quelques modifications dans les blocs 1 et 13 du contenu du fichier 'EXTEND80.BLK' afin de permettre le changement de disque en cours de compilation.

Le problème de limitation de la capacité du disque actif risque d'être évidente avec le fichier 'META80.BLK'. En effet, une fois décompressé, celui-ci fait au moins 220Koctets. Ce cas se présente nottament pour les systèmes AMSTRAD CPC dont la capacité disque est de 180Koctets au maximum. Mais la nécessité de décompresser le contenu de 'META80.BQK' n'a pas de caractère d'urgence, car le fichier 'KERNEL' résultant de la compilation de 'META80' est à votre disposition et peut être manipulé sans problème.

F83 ET 83-STANDARD

Pour démarrer FORTH depuis CP/M, il faut taper 'F83', et sous MSDOS 'RUNME'. Pour les systèmes AMSTRAD CPC, il faut d'abord taper 'CPM' précédé de la barre verticale pour passer sous CP/M. N'oubliez pas de charger au préalable votre disquette système. Au démarrage, FORTH se présente, sous CP/M, en affichant:

```
8080 Forth 83 Model 1.0.0
Modified 16Oct83
```

Pour taper votre première commande, passer en mode majuscules, puis essayez 'WORDS', ce qui correspond au 'VLIST' des standards FIG et 79-STANDARD. L'appui sur une touche quelconque interrompt l'édition du contenu du dictionnaire.

A première vue, ce vocabulaire est très riche. Pour visualiser le contenu des autres vocabulaires, il faut taper le nom du vocabulaire concerné suivi de 'WORDS'. Les noms des différents vocabulaires sont listés par la commande

→

'VOCS', ce vous affiche:

SHADOW EDITOR HIDDEN BUG FILES CP/M ONLY USER ASSEMBLER

Ces vocabulaires contiennent des utilitaires spécifiques à certaines tâches, telle l'édition, la gestion des écrans commentaires associés, le décompilateur, le débuggeur, l'accès au disque, etc...

Si vous avez déjà une certaine expérience des systèmes FORTH au standard FIG ou 79-STANDARD, vous aurez l'impression de vous sentir un peu perdu dans ce nouveau système qui ne compte pas loin de mille mots, tous vocabulaires confondus. En fait, cette richesse est dûe au souci de portabilité maximale qui animait les créateurs de ce système. Le système 83-STANDARD, tel qu'il est défini par le 'FORTH STANDARD TEAM', est beaucoup plus dépouillé. Le standard ne précise que la manière dont un mot doit s'executer. Par conséquent, un mot standard peut avoir une définition incluant des mots non standards, pouvu que son fonctionnement soit standard. Prenons un exemple: le mot 'EMIT'.

Le mot 'EMIT' est un mot vectorisé, c'est à dire que sa définition peut se résumer pour sa partie exécution à une définition du type:

```
: EMIT  EXECUTE  ;
```

où 'EXECUTE' exécute le cfa du mot situé au sommet de la pile de données. En fait, 'EMIT' a été défini de la manière suivante:

```
DEFER EMIT
```

L'exécution de 'EMIT' n'est pas encore initialisée. Pour ce faire, il faut lui attribuer un vecteur qui peut être une primitive définie avant ou après le mot vectorisé 'EMIT'. Dans le cas de 'EMIT', trois primitives sont disponibles, '(EMIT)', '(PRINT)' et '(PEMIT)'. L'affectation d'une de ces primitives en tant que vecteur au mot 'EMIT' est réalisée de la manière suivante:

```
' (EMIT) IS EMIT
```

Chacune de ces primitives n'est pas standard. Seule 'EMIT' est standard. Afin de mieux vous pénétrer des implications profondes de ce qui précède, essayez ce qui suit:

```
: (DEMIT)  DUP (EMIT) (EMIT)  ;
' (DEMIT) IS EMIT
```

Et maintenant, ne soyez pas SSUURRPPRRIISS par ce qui se passe à l'affichage. Pour rétablir une situation normale, tapez:

```
' (EMIT) IS EMIT
```

ce qui apparaît à l'affichage sous la forme

'' ((EEMMIITT)) IISS EEMMIIT (ne vous inquiétez pas, ça lui passera dès que vous aurez validé par appui sur la touche RETURN).

Il en est ainsi pour beaucoup de fonctions dont 'KEY', 'CR', 'KEY?', 'CHAR', 'DEL-IN', 'LOAD', 'NUMBER', etc... Ces différents vecteurs dépendants eux-même de paramètres non standards et qui auraient pu être exprimés sous forme littérale au sein des définitions standards.

Voici la liste des différents vecteurs et les définitions exécutées par défaut définis dans F83 sous CP/M:

vecteur:	mot exécuté:
EMIT	(EMIT)
KEY?	(KEY?)
KEY	(KEY)
CR	CRLF
CHAR	(CHAR)
DEL-IN	(DEL IN)
READ-BLOCK	FILE-READ
WRITE-BLOCK	FILE-WRITE
LOAD	(LOAD)
NUMBER	(NUMBER)
SOURCE	(SOURCE)
STATUS	CR CRLF
WHERE	NOOP
?ERROR	(?ERROR)
BOOT	HELLO
CONVEY-COPY	(COPY)
AT	(AT)
BLOT	(BLOT)
-LINE	NOOP
INIT-PR	NOOP

Les vecteurs sont identiques sur RUNME sous MSDOS.

Dans d'autres cas, la partie exécution du mot est découpée en plusieurs sous-fonctions. C'est le cas du mot 'DUMP', lequel exécute les mots '.2', 'D.2', 'EMIT.', 'DLN', '?.N', '?.A', et '.HEAD'. Par conséquent, ne cherchez pas à connaître le fonctionnement de tous les mots apparaissant à l'exécution de 'WORDS', seuls comptent ceux définis par le 83-STANDARD.

LE DECOMPILATEUR

Nous avions dit précédemment que cette version FORTH 83-Standard est un véritable système de développement. Le programme F83 inclut un décompilateur pouvant restituer la définition d'un mot à partir de sa forme compilée. Compte tenu des remarques du précédent chapitre, vous risquez cependant de naviguer dans un labyrinthe de mots non standards. La décompilation d'un mot est réalisée par la séquence 'SEE mot', où 'mot' est un mot du dictionnaire.

Le décompilateur précise la nature du mot décompilé, c'est à dire défini par:

CONSTANT	VARIABLE
:	DEFER

Dans le dernier cas, la décompilation est forcée vers la définition du mot servant de vecteur.

La décompilation restitue les structures de contrôle de type 'IF..THEN', 'IF..ELSE..THEN', 'BEGIN..UNTIL' etc... uniquement sous la forme de branchements conditionnels et inconditionnels.

LE CHAMP DE VUE

Un nouveau champ est défini dans la structure du dictionnaire, appelé 'view field address'. Ce nouveau champ est situé entre le champ de lien (lfa) et le champ du nom (nfa). Son contenu n'a aucune influence sur le système FORTH et les programmes que vous pourriez définir. Le contenu de ce champ permet simplement de pointer sur le fichier et le bloc d'origine du mot défini.

La recherche du bloc d'origine d'une définition du dictionnaire est réalisée par une séquence du type:

```
VIEW mot
```

Le bloc contenant la définition du mot succédant 'VIEW' est listé. Le fichier d'origine doit être sur le disque actif au moment de l'exécution de 'VIEW'. Pour les possesseurs de système AMSTRAD CPC vous devrez renoncer à lister par 'VIEW' un mot dont la définition est originaire du fichier 'META80'.

LE SYSTEME MULTI-TACHES

Le système F83 prévoit une utilisation multitâche (y compris sur AMSTRAD) des ressources du système hôte. En réalité, les fonctions de gestion multi-tâches se résument essentiellement à la gestion d'un tampon d'impression ou d'un compteur. Définition d'un tampon d'impression (print spooler):

→

```
BACKGROUND SPOOLER  1 CAPACITY SHOW STOP ;
```

Pour valider le spooler, taper 'MULTI'. Ce mot démarre l'exécution de la boucle multi-tâche. Pour l'arrêter, taper 'SINGLES'.

Puis taper 'SPOOLER WAKE' ce qui démarre la tâche du spooler. Pour interrompre le spooler, taper 'SPOOLER SLEEP'. Pour le redémarrer, taper à nouveau 'SPOOLER WAKE'.

Exemple d'utilisation de SPOOLER au sein d'une définition:

```
: SPOOL-THIS
  SPOOLER ACTIVATE  3 15 SHOW STOP  ;
```

L'EDITEUR

Le système F83 est livré avec un éditeur 'en ligne', ce qui rebutera nombre d'utilisateurs. Ce choix permet pourtant une très large portabilité du programme F83. Si cet éditeur ne vous convient pas, créez votre propre éditeur. Cependant, pour pouvoir entrer vos premières définitions, notez ces quelques commandes:

	P texte	insère le texte à la ligne courante
n	T	sélectionne la ligne n
+-n	C	déplace curseur +- n caractères
	TOP	met curseur ligne 0, colonne 0.
n	NEW	combine les actions de T et P avec répétition
	O texte	surimpression (Overwrite).

Avec certains systèmes, un affichage semi-plein écran peut être obtenu en choisissant une des options terminal HEAT, FALCO, TELEVIDEO, QUME, ANSI, PERKIN. Si aucune de ces options ne donne satisfaction, vous pouvez revenir à la situation initiale en tapant DUMB.

LES FICHIERS

Autre nouveauté, les fichiers 83-Standard du Forth conçu par Laxen et Perry gère des fichiers compatibles avec le système d'exploitation résident. De plus, les programmes sources élaborés sous CP/M sont téléchargeables et compatibles sous MSDOS et inversement. Cette compatibilité se réduit aux seules commandes standard. Les définitions écrites en assembleur ne sont compatibles qu'avec un micro-processseur de même type et sur un système ayant les mêmes caractéristiques.

N'importe quel fichier peut être ouvert, y compris les fichiers générés par d'autres langages (Turbo-PASCAL, BASIC, dBASE II, LOGO, WORDSTAR, etc...) à charge pour vous de définir les routines capables de les exploiter. Exemple:

- sous Turbo-PASCAL, générez un fichier quelconque nommé 'fichier.PAS'
- sous F83, tapez 'OPEN fichier.PAS'
- 1 LIST affichera une partie du contenu de 'fichier.PAS'.

Dans le cas présent, le listage est possible, car 'fichier.PAS' est au format ASCII. Mais dans le cas de fichiers de données issues d'un programme de type dBASE II ou d'un fichier binaire, ce listage ne sera pas possible directement.

Pour créer son propre fichier source, il faut taper:

```
n CREATE-FILE fichier.BLK
```

où n indique le nombre de blocs écrans prévus dans votre fichier. Un conseil, si votre fichier est un fichier de travail, prévoyez large. Une fois votre frappe terminée, vous pourrez toujours le recopier en le raccourcissant à l'aide de la commande PIP disponible sous CP/M.

Vous pouvez visualiser les noms de fichiers disponibles en tapant 'DIR' sous FORTH. L'ouverture d'un fichier génère un en-tête dans le dictionnaire de libellé identique au nom du fichier ouvert. Pour réouvrir par la suite ce même fichier, il suffira de retaper simplement le nom du fichier. Exemple:

```
- ouverture:    OPEN fichier1.ext
- ouverture:    OPEN fichier2.ext
- réouverture:       fichier1.ext
```

L'activation du mode éditeur est réalisée par la commande 'n EDIT'. Le système FORTH demande votre cachet d'identification. Ce cachet permet de personnaliser le contenu des blocs écrans sources. Par convention, un cachet d'identification type comprend:

- 2 chiffres pour le numéro du jour
- 3 lettres pour le nom du mois en abrégé: JAN, FEV, MAR, etc...
- 2 chiffres pour le millésime
- 2 à 3 caractères pour les initiales du programmeur.

Exemple: 05sep86 MP

Ce cachet d'identification est apposé à l'extrémité de la ligne 0 des blocs écrans mis à jours. L'abandon d'une session d'édition avec mise à jour des fichiers est réalisée par 'DONE'.

Une documentation complète est en cours d'élaboration, reprenant l'ensemble des différents vocabulaires. Ce glossaire sera disponible dans quelques semaines. En attendant, vous pouvez faire part de vos questions et découvertes en appelant Marc PETREMANN au 46.56.33.67 hdb.

suite de la page 5

```
 0 ( EXECUTION VECTORISEE : STRUCTURE DE CASE )
 1
 2 : CASE: CREATE         ( crée une en-tête dans le dic)
 3       SMUDGE           ( valide l'en-tête )
 4       ]                ( compile cfa mots qui suivent)
 5       DOES>            ( partie exécution )
 6       SWAP 2* +        ( calcule l'adr contenant cfa)
 7       @EXECUTE ;       ( execution si cfa diff. Ø)
 8
 9 CASE:  ETAT-CIVIL    MR ME ML EN ;
10
```

```
 0 ( MACHINE A CALCULER )
 1
 2 VARIABLE COMPTEUR
 3 VARIABLE COMPTE-SOMMES
 4 VARIABLE DERNIERE-SOMME
 5
 6 : REPETE  ( n - )
 7    COMPTEUR 1 OVER +! @
 8    5 SPACES 2 .R
 9    DUP DERNIERE-SOMME !
10    DERNIERE-SOMME @ 10 .R
11    COMPTE-SOMMES +!
12    COMPTE-SOMMES @ 10 .R ;
13
```

```
 0 ( MACHINE A CALCULER )
 1
 2 : MM ( n - )
 3    O COMPTEUR !
 4    O COMPTE-SOMMES !
 5    O DERNIERE-SOMME !
 6    BEGIN CR INPUT ?DUP
 7      IF REPETE
 8      ELSE ." Total...: " COMPTEUR @
 9           5 SPACES 2 .R O 10 .R
10           COMPTE-SOMMES @ 10 .R QUIT
11      THEN
12    O UNTIL ;
13
```

Suite page 16

SIGNATURE ET HASHCODE

Le problème typique est de transformer une chaîne de caractères, de longueur variable, et dont la capacité de codage, c'est à dire le nombre total de chaînes différentes,est astronomique, en un nombre entier compris entre deux bornes, appelé signature.

S'il y a N objets à ranger en C cases, il y aura, en général, collision: plusieurs objets dans la même case, et manque: aucun objet dans une case.

- Deux objets de même nom ont la même image, toutefois l'inverse est faux.
- Si l'image manque, l'objet n'a pas encore été classé.

Les applications sont multiples:

- accélérer les recherches dans un dictionnaire: pour chercher N objets en R essais, il faut N/(2*R) pointeurs d'entrée en dictionnaire, plus un pointeur par objet, chaînant les mots de même signature.
- détecter les erreurs: la signature est ajoutée au texte avant transmission, archivage ou frappe, puis contrôlée. Le risque de ne pas voir l'erreur est de 1/G. G: nombre de signatures valides.

Le calcul d'une signature est généralement basé sur une suite pseudo-aléatoire.

LES SUITES PSEUDO-ALEATOIRES

La suite: $u_{n+1} = u_n*A + B$ @ G (modulo G) a une période au plus égale à G.

Pour G impair, elle a, en général:

- une valeur unique u_o invariante
- une suite de période G-1, ou un de ses sous-multiples.

S'il n'y a pas de sous-période, G-1 itérations feront apparaître chaque valeur une seule fois, sauf u_o: aucune collision, un seul manque.

En général, on rajoute une valeur numérique c dépendant du caractère examiné, à la suite précédente: $u_{n+1} = u_n*A + B + c$ @G

Pour A = 1 , B = 0 , c = n° de code ASCII, la transformation s'appelle CHECK-SUM.

PROBLEMES DE HASHCODE

1) Pour N objets et G cases, combien de vides ?
2) Pour N bien supérieur à G, remplissage minimum, moyen et maximum de chaque case ?

1) La solution du premier est très simple: probabilité qu'une case donnée ne soit pas touchée lors de 1 tirage : p = 1 -1/G

N tirages: p_N= p**N

p_N= e** (N *$\log_e$(1 -1/G)) ≃ e**(-N/G)

Pour 300 objets et 100 cases:

p= 1 -1/100 = 0,99 -$\log_e$p= 0,01005033

U= -N* log p = 3,0151 = I* log 2 +b
= 4* 0,693 +0,2431

e**x - 1+ (2*x) /(2-x) à 0,3% près pour |x| <0,35

e**0,2431 ≃ 1+ 2/1,7569 =1,2767

U= 2**I * e**b = 2**4 * e**0,2431 = 16 * 1,2767
= 20,43

Lors de multiples essais, ce nombre serait réparti selon une loi non détaillée ici, mais dont la moyenne serait 20,43.

2) Avant de résoudre le deuxième problème, il parait nécessaire de faire un détour par le calcul des probabilités. Je sais, ce n'est pas très digeste, mais mieux vaut, si nécessaire,se prendre la tête entre les mains pour bien comprendre, et après tout (ou presque) devient facile!

CALCUL DE PROBABILITE

Appelons épreuve un processus parfaitement défini, qui fournit une détermination x_i de la variable aléatoire x, par exemple:

- le jet d'un dé
- la mesure physique d'une constante
- la longueur d'un rail pour une quantité donnée de métal
- etc...

S'il n'y apas d'effet de mémoire, le processus est entièrement caractérisé par sa loi de distribution, c'est à dire la proportion de cas observés entre x et x+Δx, ou son intégrale, la loi de répartition: probabilité que x soit inférieur à x1 donné.

Petite digression: voulez-vous générer des nombres aléatoires répartis selon une loi de distribution définie à l'avance, par exemple gaussienne? Rien de plus simple: son intégrale P(x) varie de 0 à 1. Il suffit de créer une fonction (pseudo) aléatoire y de distribution uniforme de 0 à 1, chaque valeur x_i est donnée par: $P(x_i)= y_i$.

N épreuves fournissent N valeurs x_i, chacune d'elles est distribuée autour d'une moyenne $\overline{x}$, le carré de l'écart à cette moyenne s'appelle la variance v.Leur somme a les propriétés suivantes:

- la moyenne de la somme est égale à $N*\overline{x}$
- la variance de la somme est égale à la somme des variances élémentaires
- la somme converge vers une loi de distribution gaussienne, caractérisée par sa moyenne et son écart type.

Le calcul peut s'effectuer en trois temps:

- noter les valeurs
- calculer la moyenne
- calculer la variance moyenne,

ce qui nécessite une case mémoire par mesure, ou également en deux temps. A chaque mesure, on actualise 3 accumulateurs:



MO:= MO +1   nombre de mesures
M1:= M1 $+x_i$ somme des mesures
M2:= M2 $+x_i$**2 somme des carrés
Et à la fin:
Nombre   N := MO
Moyenne: x_o:= M1/N
Variance: v:= M2/N -(x_o**2)

LOI GAUSSIENNE

Pour ceux qui veulent effctuer les calculs, voici les formules:

Fonction de distribution:

$$err(x)= Z(x)= \frac{1}{\sqrt{2\pi}} * e^{-\frac{x^2}{2}}$$

Ses trois premiers moments sont:
MO= somme de   err(x) =1
M1=   x*err(x) =0 (fonction symétrique)
M2=   x**2 *err(x) =1 (variance)

Fonction de répartition:

Probabilité qu'une variable soit inférieure à x: P(x)

Fonction centrée de répartition:

Probabilité qu'une variable soit comprise entre -x et +x :

$$erf(x)=A(x)=\sqrt{\frac{2}{\pi}}*(x \quad \frac{x^3}{1!3} + \frac{x^5}{2!5} \quad \frac{x^7}{3!7} \quad +..)$$

$$A(x)= 1 -\sqrt{\frac{2}{\pi}} *\frac{e^{-\frac{x^2}{2}}}{x} *(1 -\frac{1}{x^2} +\frac{1*3}{x^4} - \frac{1*3*5}{x^6} + ..)$$

Il faut arrêter le calcul de la deuxième formule quand l'oscillation est minimale.

P et A sont reliées par:

$$2 * P(x) = 1 +A(x)$$

A(x) a les valeurs suivantes:

±x	0	0.6845	1	1.1498	1.645	1.96	2
A(x) %	0	50 quartile	68.268 écart type	75	90	95	95.45

±x	2.576	3	3.286	3.890	4	4.417
A(x) %	99	99.730	99.9	99.99	99.9968	99.999

±x	4.893	5.32	1-A(x)	5	6	7
A(x) %	99.9999	99.99999	(log_{10})	6.2416	8.704	11.59

Revenons à nos moutons: je fais un tirage et, pour une case donnée, je compte 0 si elle n'est pas touchée, et 1 dans le cas contraire. Ceci est une variable aléatoire, de moyenne   p= 1/G $=\overline{x}$
la moyenne des carrés des écarts à 0 est aussi de $\overline{x^2}$= 1/G
et la variance: $v=\overline{(x-\overline{x})^2} = \frac{1}{G} - \frac{1}{G^2} = \frac{G-1}{G^2}$

Pour la somme des N tirages, on obtient:
Moyenne: $\overline{S}$= N/G
Variance: V= N*v = $N*\frac{G-1}{G^2}$
Ecart type:
racine carrée, soit $\sqrt{S}$ si G est grand.

Pour le cas très important où G=2 (pile ou face): $\overline{x}$= 1/2;   v= 1/4 ;
$\overline{S}$= N/2;   V= N/4 ; $=\sqrt{N}$ / 2 ;

Pour une série de 10000 tirages pile ou face, la somme des pile sera de 5000 avec un écart type de 50 .

Pour N= 800 et G= 200, on obtient:
remplissage moyen $\overline{S}$= N/G =4
écart type $\sqrt{4}$ = 2

Les valeurs extrêmes, qui n'ont que 1/G =1/200 chance d'être dépassées sont:
4± 2* 2.576 = 4± 5.152
(P(x) = 0.995, A(x) = 0.99, x= ± 2.576)

La valeur minimale est négative: il y aura plusieurs cases vides, combien
x= $\overline{S}$ /σ = $\sqrt{S}$ = 2 P(x)= (1-0.9545)/2 =0.02275
G*P(x)= 4.55

Autre calcul possible:
probabilité qu'une case reste vide:
(1-1/G)**N ≃ e**(-N/G) = e**(-4) = 1/54.6
nombre de cases vides: G/54.6 = 3.66

L'écart entre ces résultats provient du caractère sommaire de la théorie et non des approximations faites.

Un résultat, qui pourrait paraître curieux a été établi: la proportion de cases vides ne dépend que du remplissage moyen, il est de 2% pour un remplissage de 4. ■

```
0 ( ENTREES ET SORTIES )
1
2 : SIGNE
3    2DUP XOR
4    >R ABS >R
5    DABS
6    R> R> ;
7
8 : D*
9    SIGNE >R
10   DUP 2SWAP
11   ROT *
12   ROT ROT U*
13   ROT 0 SWAP
14   D+ R>
15   D+- ;
```

```
0 ( ENTREES ET SORTIES )
1
2 : INPUT   PAD 10 EXPECT PAD 1- NUMBER ;
3
4 : $IN    INPUT
5    DPL @ 1 < IF 100 D* ELSE
6    DPL @ 1 = IF  10 D* ELSE
7    DPL @ 2 = IF   1 D* ELSE
8    ." SVP 2 CHIFFRES SEULEMENT APRES LA VIRGULE "
9    2DROP  THEN THEN THEN ;
10
11 : F    >R SWAP OVER DABS
12   <# # # 46 HOLD #S SIGN #>
13   R> OVER - SPACES
14   TYPE SPACE ;
15
```

```
0 ( ENTREES ET SORTIES ).
1
2 : DD.   SWAP OVER DABS
3   <# DPL @ DUP 0=
4      IF DROP D. DROP
5      ELSE 0 DO # LOOP
6         46 HOLD #S SIGN #>
7         TYPE SPACE
8      THEN ;
9
```
■

PROLOG Les opérations arithmétiques

par Marc PETREMANN

A l'origine, Prolog ne semble guère doué pour le calcul numérique. Il n'en est rien. Tout système Prolog digne d'intérêt dispose de prédicats autorisant le traitement des valeurs numériques selon les quatres opérations élémentaires.

Considérons la somme de deux nombres entiers a et b, le résultat c sera vérifié si a+b=c, opération notée en Prolog sous la forme:

```
?:+(5,7,12).
-- SUCCES --(1)--
```

Si le résultat doit être calculé, on remplacera le troisième argument par une variable non instanciée:

```
?:+(5,7,*X).
*X      = 12
-- SUCCES --(1)--
```

On peut procéder à plusieurs calculs simultanément, ceci dans la limite de la chaîne de caractères admise en entrée par Prolog:

```
?:+(5,7,*X)&+(*X,10,*Y).
*X      = 12
*Y      = 22
-- SUCCES --(1)--
```

La résolution du premier prédicat instancie la variable *X, ce qui permet la résolution du second prédicat. On ne peut résoudre un prédicat d'opération sans avoir instancié les deux premiers termes ou les avoir définis en tant que constantes:

```
?:+(*X,6,12).
ERR 1 :EXECUTION
```

Pour pouvoir déterminer la valeur d'une variable quelle que soit sa place dans le prédicat, il faudra définir les règles de résolution correspondantes:

```
a+b=x   est résolu par   x=a+b
a+x=c   est résolu par   x=c-a
x+b=c   est résolu par   x=c-b
```

A partir de là, on définit les règles suivantes:

```
=1=<11>SOM(*X0,*X1,*X2):
      VAR(*X2)&
      +(*X0,*X1,*X2)&
      CUT.
=2=<11>SOM(*X0,*X1,*X2):
      VAR(*X1)&
      -(*X2,*X0,*X1)&
      CUT.
=3=<11>SOM(*X0,*X1,*X2):
      VAR(*X0)&
      -(*X2,*X1,*X0)&
      CUT.
```

Et dont on peut vérifier le fonctionnement:

```
?:SOM(2,*X,7).
*X      = 5
-- SUCCES --(1)--

?:SOM(*X,5,7).
*X      = 2
-- SUCCES --(1)--
```

On peut étendre ce principe aux trois autres opérations arithmétiques:

```
=1=<11>DIF(*X0,*X1,*X2):
      VAR(*X2)&
      -(*X0,*X1,*X2)&
      CUT.
=2=<11>DIF(*X0,*X1,*X2):
      VAR(*X1)&
      -(*X0,*X2,*X1)&
      CUT.
=3=<11>DIF(*X0,*X1,*X2):
      VAR(*X0)&
      +(*X1,*X2,*X0)&
      CUT.
```

```
=1=<11>PROD(*X0,*X1,*X2):
      VAR(*X2)&
      *(*X0,*X1,*X2)&
      CUT.
=2=<11>PROD(*X0,*X1,*X2):
      VAR(*X1)&
      DIV(*X2,*X0,*X1)&
      CUT.
=3=<11>PROD(*X0,*X1,*X2):
      VAR(*X0)&
      DIV(*X2,*X1,*X0)&
      CUT.

=1=<11>QUOT(*X0,*X1,*X2):
      VAR(*X2)&
      DIV(*X0,*X1,*X2)&
      CUT.
=2=<11>QUOT(*X0,*X1,*X2):
      VAR(*X1)&
      DIV(*X0,*X2,*X1)&
      CUT.
=3=<11>QUOT(*X0,*X1,*X2):
      VAR(*X0)&
      *(*X2,*X1,*X0)&
      CUT.
```

Et maintenant nous sommes prêts à résoudre n'importe quelle équation du premier degré à une inconnue, ceci quelle que soit la position de l'inconnue dans notre équation:

```
(3*X)+5=26
```

sera décomposé en deux opérations élémentaires:

```
( 3 * X ) + 5 = 26
 -----     -
   !       !
  N2       !
 ----------
      !
      N1
```

et s'exprime en Prolog par:

```
?:SOM(*Y,5,26)&PROD(3,*X,*Y).
```

Suivons la résolution en activant le mode "trace" par la commande TRON (sur Prolog FIL pour THOMSON):

```
     --> SOM(*X0,5,26)
=1= SOM(*X0,5,26)
     --> VAR(26)
     <-- SOM(*X0,5,26)
=2= SOM(*X0,5,26)
     --> VAR(5)
     <-- SOM(*X0,5,26)
=3= SOM(*X0,5,26)
     --> VAR(*X0)
     --> -(26,5,*X0)
     --> CUT
     --> PROD(3,*X1,21)
=1= PROD(3,*X1,21)
     --> VAR(21)
     <-- PROD(3,*X1,21)
=2= PROD(3,*X1,21)
     --> VAR(*X1)
     --> DIV(21,3,*X1)
     --> CUT
*Y      = 21
*X      = 7
-- SUCCES --( 1 )--
```

CONCLUSION: Prolog permet le traitement des opérations arithmétiques élémentaires, et, par application de règles simples, d'étendre son pouvoir de résolution à ce domaine un peu trop fréquemment oublié de la littérature concernant les systèmes experts. En fait, on peut considérer que l'ensemble des règles mathématiques puissent constituer une base de connaissance applicable à la résolution de problèmes plus complexes que celui précédemment traité. C'est le cas de Mumath, disponible sur Apple II, écrit en LISP, qui traite les opérations mathématiques sous forme symbolique et, éventuellement numérique. A noter aussi, certains programmes LOGO diffusés dans le défunt magazine de l'AFUL, permettant d'opérer des simplifications d'équations.

■

Le PCW est doté dès sa naissance d'une belle bibliothèque de programmes intéressants puisqu'elle compte notamment MULTIPLAN, dBASE II et TURBO PASCAL. A ceux-ci vient de s'ajouter un compilateur C fort intéressant: le HISOFT C, Hisoft étant une société spécialisée dans les outils de programmation.

Ce compilateur reprend la même philosophie que TURBO PASCAL (est-ce un hasard!...), c'est à dire une très grande rapidité de compilation et production directe d'un module exécutable sans passer par la phase fastidieuse du 'link' (édition des liens). De plus, il reprend le quasi-standard créé par Kernighan et Ritchie (à l'exception toutefois des flottants), ce qui permet d'accéder à de nombreux sources du domaine public.

Par ailleurs, il est livré avec une documentation très complète sous forme de classeur et, ce qui ne gâche rien, avec un excellent éditeur de texte orienté programmation.

Ce compilateur se veut "UNIX-LIKE", c'est à dire qu'il se rapproche autant que faire se peut de l'utilisation sous UNIX. Néanmoins, l'utilisation sous CP/M pose certains problèmes, notamment au niveau de la gestion des entrées-sorties et du passage d'arguments à l'appel du programme depuis CP/M. Conscients de ce problème, les développeurs de Hisoft livrent avec le compilateur une librairie appelée CPM.LIB. Celle-ci donne accès à de nombreuses fonctions de CP/M depuis le C.

```
/****************************************************************/
/*                                                              */
/*      COMPTE LE NOMBRE DE PARENTHESES DANS UN SOURCE          */
/*      -----------------------------------------------         */
/*                                                              */
/*      Gilles BERTIN , a CHALON/SAONE le 6 avril 1986          */
/*      avec le HISOFT-C sous CP/M 3.0 sur AMSTRAD PCW          */
/*                                                              */
/****************************************************************/


#include stdio.h

main(argc,argv)
int argc;               /* contient le nombre de chaines passees */
                        /* y compris le nom du programme */
char *argv[];           /* definit une chaine de pointeurs, pointeurs */
                        /* qui pointent sur des chaines de caracteres */

{
  int  car,ouvre1,ferme1,ouvre2,ferme2;
  FILE *fichier;
  FAST char argv_buffer[MAXARGS*2+132];   /* voir pages 115-116 HISOFT-C */

  ouvre1=0; ferme1=0; ouvre2=0; ferme2=0;
  cpm_cmd_line(&argc,&argv,argv_buffer);  /* simule passage de commande sous UNIX*/
  if (--argc == 1)
         {
         fichier=fopen(*++argv,"r");      /* ouvre le fichier demande */
         if (fichier!=0)
              {
              for ( ; (car=getc(fichier)) != EOF; )
                   {
                   switch (car)
                        {
                        case '{' : ++ouvre1; break;
                        case '}' : ++ferme1; break;
                        case '(' : ++ouvre2; break;
                        case ')' : ++ferme2; break;
                        default  : break;
                        }
                   }
              printf("Nombre de { = %d\n",ouvre1);
              printf("Nombre de } = %d\n",ferme1);
              printf("Nombre de ( = %d\n",ouvre2);
              printf("Nombre de ) = %d\n",ferme2);
              }
         else
              { printf("Le fichier n'existe pas sur ce disque.",*argv); }
         }
    else
         {
         if (argc==0) printf("Donnez un nom de fichier.");
               else printf("Trop d'arguments. %d\n",argc);
         }
}

#include ?cpm.lib?      /* ne compile que les fonctions necessaires */
#include ?stdio.lib?    /* au programme */
```

AMSTRAD PCW

Le programme ci-joint montre comment passer des arguments de CP/M à une application C. Son but est de compter les parenthèses ouvrantes et fermantes dans un source C, celles-ci devant être en nombre égal...

Il s'agit donc de passer à ce programme le nom du source. il faut pour cela utiliser la procédure CPM_CMD_LINE donnée dans CPM.LIB et créer un buffer permettant le stockage des données passées. Par la même occasion, le nom du programme est lui-même passé. On peut ainsi créer un programme qui vérifie son propre nom et qui refuse de fonctionner si ce nom a été changé!

On peut ensuite utiliser ces données de manière classique. ARGC contient le nombre d'arguments passés, y compris le nom du programme, et ARGV est un pointeur pointant dans un tableau de pointeurs, ces derniers pointant sur les chaînes de caractères arguments. A remarquer dans ce programme le FOPEN(*++argv,"r"). Il convient en effet de bien pointer sur le second argument, le premier étant le nom du programme; le --argc est là) pour la même raison.

Dernière remarque: les deux derniers INCLUDE sont conditionnels, c'est à dire que le compilateur ne va chercher dans les fichiers cités en INCLUDE que les procédures dont il a besoin. ■